



AI Agents & Small AI Apps

THE VISUAL STUDIO LEARN STARTER PACK

Build your first AI agent, package its tools the standard way, and ship a complete PDF Q&A app — the three flagship tutorials, distilled into one free guide.

Every number in this pack is real: the live agent run, the actual MCP tool output, and a PDF answer that cost US\$0.000339 — a thirtieth of a cent. No framework magic. The loop, the protocol, and the pipeline, explained plainly.

Part 1 · The Agent Loop — an agent is a prompt, plus tools, plus a loop

Part 2 · MCP in Plain Terms — one standard way to give agents tools

Part 3 · Small AI App #1 — a PDF Q&A app with FastAPI + Gemini

Plus · the run-everything setup checklist

Visual Studio Learn · visualstudiolearn.blogspot.com

Practical tutorials on Python, FastAPI, Microsoft Azure, SQL Server, and AI agents — new posts and videos every Tuesday, Thursday, and Saturday.

Contents

How to use this pack — read it in order, or jump to the app

Part 1 — The Agent Loop · the idea · the two tools · the safety rule · the live run · 4 errors checklist

Part 2 — MCP in Plain Terms · the wiring problem · Host / Client / Server · a tool is a typed function · the live result · key errors

Part 3 — Small AI App #1: PDF Q&A · the pipeline · cost per answer · the errors table

Run everything — one setup checklist, versions pinned

Where to go next — the full tutorials, the videos, the code, and new posts by email

How to use this pack

Each part distils one complete, published Visual Studio Learn tutorial — the same code that ran on the blog and in the videos, with the same real results. Read the parts in order and you get the whole arc: how an agent works, how its tools get packaged so any agent can use them, and how both ideas become a small app you can run in under an hour. Every part ends with the errors you will actually hit, because those are the pages everyone eventually needs.

The full code for all three builds lives in one public repository — github.com/sajjad037/visual-studio-learn-code — one folder per tutorial, exactly as printed in the posts. Links to everything are on the last page.

PART 1

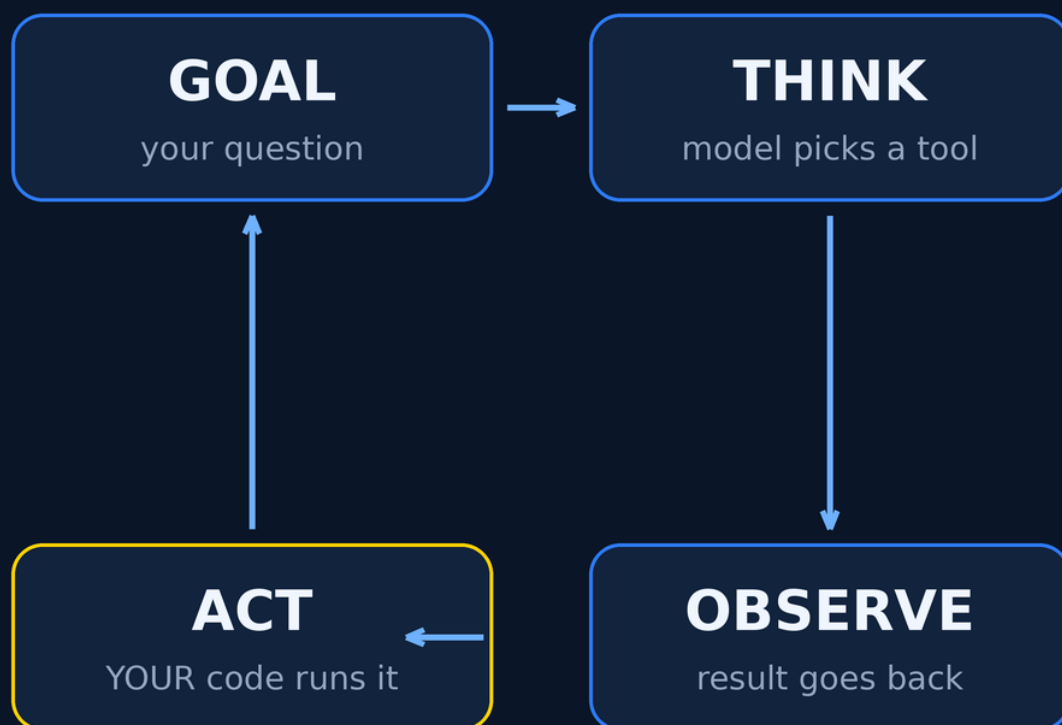
The Agent Loop

Build Your First AI Agent in Python — the tutorial, distilled

The one idea

An AI agent is not a special kind of model. It is a normal program that does three things in a loop: it sends your question and a list of tools to a model, it runs whichever tool the model asks for, and it sends the tool result back so the model can answer — or ask for another tool. Execution stays with your program from start to finish. The model only ever asks.

That is why the tutorial builds the loop by hand, with automatic function calling switched off. Frameworks hide the loop — and the loop is the part you need at two in the morning when something breaks in production.



No tool call in the reply? That reply IS the answer — the loop stops.

Round by round

- **Goal.** You send the question plus the tool list. The model sees function names, arguments, and docstrings — never the function bodies.
- **Think.** The model replies with a request, not an answer: “call this tool, with these arguments.” Nothing has run yet.
- **Act.** Your code validates the request — the tool name, the arguments — and only then executes. This is your security boundary.
- **Observe.** The result goes back into the conversation, so the model reasons over real numbers instead of inventing them. A reply with no tool call is the final answer; the loop stops.
- **A step cap protects you.** The tutorial uses `MAX_STEPS = 6`. An agent without a cap can loop — and bill you — for a very long time.

The two tools

A tool is just a Python function with a clear name, typed arguments, and a short docstring. The docstring matters more than usual: the model reads it to decide when the tool applies, so write it for the model, not only for humans.

Tool 1 — Weather lookup

Gets the current temperature and wind for a city from the free Open-Meteo service — no API key needed. Given “Ottawa”, it geocodes the name to coordinates, then asks Open-Meteo for the current conditions at that point.

Tool 2 — A calculator that refuses to be dangerous

The lazy version of a calculator tool is one line: pass the expression to Python's `eval()`. Do not do that. `eval()` runs whatever it is given, and what it is given comes from a model. The tutorial's calculator instead parses the expression into a tree and walks it, allowing only numbers and five operators — add, subtract, multiply, divide, negate. Everything else is rejected:

```
_OPERATORS = {
    ast.Add: operator.add,    ast.Sub: operator.sub,
    ast.Mult: operator.mul,  ast.Div: operator.truediv,
    ast.USub: operator.neg,
}

def _eval_node(node):
    if isinstance(node, ast.Constant):
        return node.value      # a number: allowed
    if isinstance(node, ast.BinOp)
        and type(node.op) in _OPERATORS:
        return _OPERATORS[type(node.op)](
            _eval_node(node.left), _eval_node(node.right))
    raise ValueError("Not allowed") # anything else: rejected
```

THE SAFETY RULE — say it as a rule, because it applies to every agent you will ever build: treat every tool argument as untrusted input. Because it is.

Both tools share one more discipline: they return errors as data — an `{"error": ...}` result — instead of crashing. The agent gets a chance to recover, and the conversation does not die on one bad call.

The live run

The tutorial's test question needs both tools: *“Is it warmer in Ottawa or Toronto right now, and what is the temperature difference times two?”* Here is the run, exactly as printed:

```
step 1: get_weather({'city': 'Ottawa'})
-> {'city': 'Ottawa', 'temperature_c': 9.4, ...}
step 1: get_weather({'city': 'Toronto'})
-> {'city': 'Toronto', 'temperature_c': 12.1, ...}
step 2: calculate({'expression': '(12.1 - 9.4) * 2'})
-> {'result': 5.4}
```

Answer: Toronto is warmer right now, at 12.1°C compared with 9.4°C in Ottawa – a difference of 2.7 degrees, which doubled is 5.4.

Every number in that answer came from a tool, not from the model's memory. Two weather calls in step one, the calculator in step two working on the real temperatures it just received, then the answer.

What one run costs

A two-city question typically uses roughly 1,000–1,500 input tokens and 150–250 output tokens across all its model calls. At the published gemini-2.5-flash paid rates (US\$0.30 per million input, US\$2.50 per million output), that is well under a tenth of a cent per question; on the free tier it counts against quota instead. The real cost risk is never one expensive call — it is a loop that quietly runs forty cheap ones. Log `response.usage_metadata` on every step, from day one.

The 4 errors checklist

- **KeyError: 'GEMINI_API_KEY' / 400 key error.** The variable is not set in the same terminal session that runs the script. Check that it exists — never print the key — and restart the terminal after setting it system-wide.
- **The model asks for a tool that does not exist.** Models invent plausible names. Look the name up in your tool dictionary and return an error result; the model usually corrects itself next step.
- **Malformed or missing arguments.** A missing city, an expression that is not arithmetic. Return errors as data so one bad call does not kill the run.
- **429 rate-limit / quota.** A loop multiplies requests — this one question costs three or four model calls. Pause and retry, lower `MAX_STEPS`, and treat 429 during testing as a quota fact, not a code bug. (A fifth, the transient 503, deserves the same patience: retry with increasing waits before changing code.)

PART 2

MCP in Plain Terms

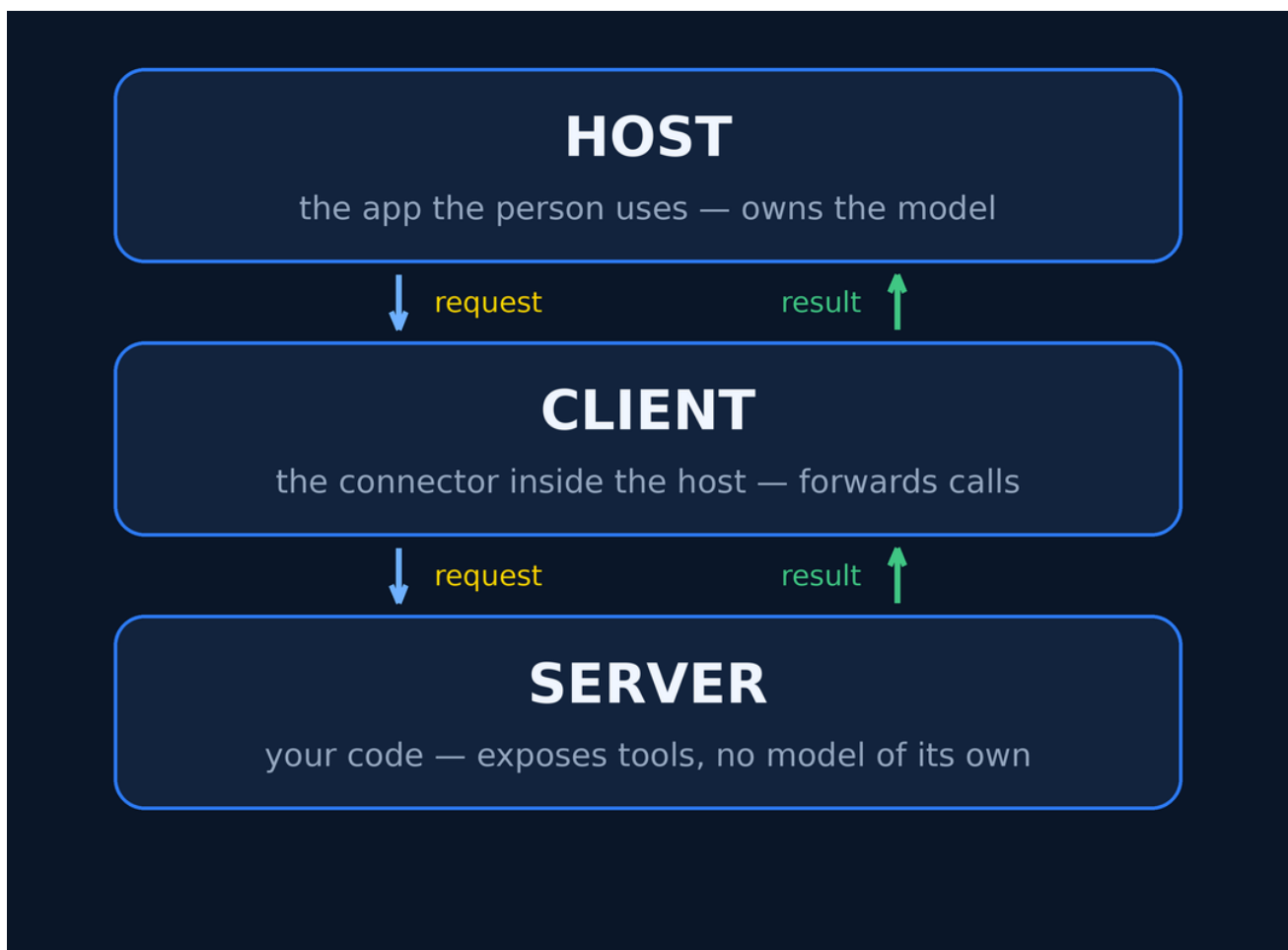
MCP Servers Explained — give your AI agent real tools, the standard way

The wiring problem

In Part 1, the tools lived inside the same file as the agent. That works — for one program. But every AI app wires its tools differently: custom glue, every single time, and a tool built for one host is useless to the next. MCP — the Model Context Protocol — is one standard protocol for exposing tools: package a tool once as an MCP server, and any MCP-aware host can list its tools, read their input schemas, and call them. Write the server once; any MCP client can use your tools.

Host, Client, Server

Three roles appear in every MCP setup, and most confusion comes from mixing them up:



- **Host** — the application the person uses: an AI coding assistant, a chat app with tool support, or your own agent program. It owns the model connection.
- **Client** — the connector inside the host, one per server. It opens the connection, performs the handshake, and forwards tool calls.
- **Server** — a separate program that offers tools and answers when called. It has no model of its own.

Client and server exchange JSON-RPC messages. The tutorial uses stdio — the server is a subprocess talking over its standard input and output: no port, no hosting, nothing exposed beyond your machine. Streamable HTTP exists for shared servers; stdio is the right start.

A tool is a typed function

The tutorial's server, project-notes, offers two read-only tools over a tiny SQLite notes database the code creates and seeds itself. Here is the whole trick — one decorated function:

```
@mcp.tool()
def search_notes(keyword: str, limit: int = 5) -> str:
    """Search project notes for a keyword
    in the title or body.

    Args:
        keyword: Word or phrase, e.g. "backup".
        limit: Maximum notes to return, 1 to 20.
    """
    limit = max(1, min(limit, 20))
    like = f"%{keyword}%"
    # ... parameterized SQL query, results as JSON
```

- **The type hints are the contract.** keyword: str and limit: int = 5 become the tool's JSON input schema automatically — no schema is written by hand.
- **The docstring is the description the model reads.** Write it as instructions to a careful colleague.
- **The SQL is parameterized.** Tool arguments are untrusted input — the Part 1 safety rule again — and binding the keyword as a parameter keeps a hostile keyword from becoming hostile SQL. Both tools are read-only, and limit is clamped.

The live result

The tutorial's client starts the server, completes the same handshake a full AI host performs, and calls both tools. Actual output, run against the MCP Python SDK 2.3.0:

```
Tools: ['search_notes', 'recent_notes']
search_notes -> {"matches": [{"id": 2,
  "title": "Database backup",
  "body": "Take a full backup before any schema
  change and verify the restore.",
  "tag": "database"}]}
recent_notes -> {"notes": [
  {"id": 4, "title": "FastAPI startup fix", ...},
  {"id": 3, "title": "Grafana alert idea", ...}]}
```

Read it as a conversation: subprocess started, handshake completed, two tools discovered, two calls answered with parseable JSON. Put a model in the host's seat and nothing about the server changes. That separation is the entire point of

the protocol — and MCP is not the model, and not the tool. It is the standard between them.

MCP — key errors

ModuleNotFoundError : 'mcp.server.fastmcp' — FastMCP was renamed	You are running v1 code on SDK v2. Change the import to <code>from mcp.server import MCPServer</code> , or pin the old line with <code>pip install "mcp<2"</code> . The generations cannot be mixed.
The server “hangs” and prints nothing	It is waiting on stdio for a client — test it with <code>mcp dev</code> or the client. And never debug-print inside the server: a <code>print()</code> on stdout corrupts the JSON-RPC stream. Log to stderr, or not at all.
A validation error names your argument	Real outputs from this server: a missing argument returns <code>Field required for keyword; "five"</code> where an integer belongs returns <code>Input should be a valid integer</code> . Both arrive as tool errors, not crashes — the caller can retry corrected.
The host cannot start the server	In order: the Python launched is not the venv's Python (use its full path); the <code>server.py</code> path is relative or wrong (absolute paths, then fully restart the host — most read this config only at startup); or the SDK sits in a different environment.

Before writing any client code, inspect the server the way a host sees it: run **mcp dev server.py**, open the Inspector page it prints, choose List Tools, and call `search_notes` with the keyword **backup**. If it works there, the server half is done — later failures live in the client or the host configuration.

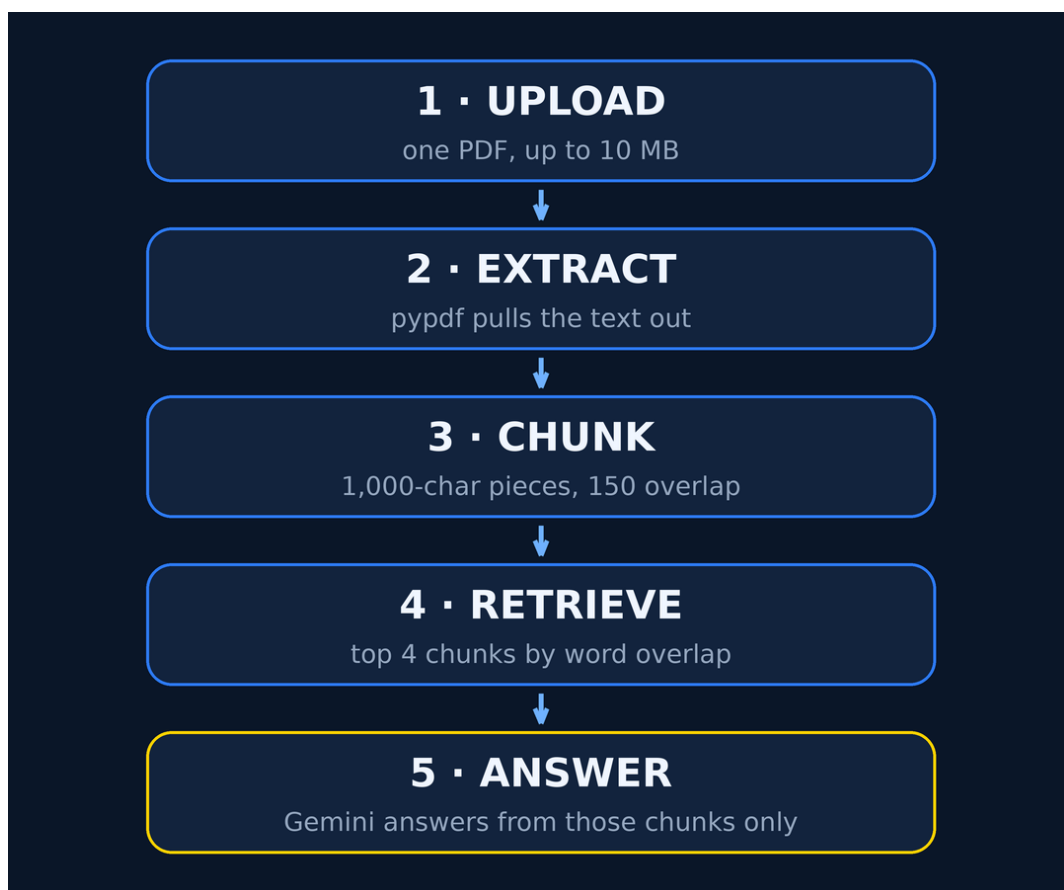
PART 3

Small AI App #1: PDF Q&A

One FastAPI file — upload a PDF, ask it questions, see what each answer cost

The whole app, honestly sized

One small FastAPI file is enough. Upload a PDF and the server extracts its text with pypdf, splits it into overlapping chunks, and stores them in memory. Ask a question and it picks the chunks sharing the most words with it, sends only those to Gemini with a strict instruction to answer from them alone, and returns the answer with the token counts the API reported and what they cost. No vector database, nothing on top of FastAPI — and if no chunk matches, the app does not call the model at all.



- **POST /upload** — accepts one PDF, extracts and chunks its text, returns a document id plus the real page, character, and chunk counts.
- **POST /ask** — answers one question from the best-matching chunks, and reports chunks used, input tokens, output tokens, and estimated cost.

- **GET /** — a one-page web UI: pick a file, upload it, ask questions, read the cost line under every answer.
- **GET /health** — a liveness check, because every app you run should have one.

Two honest design choices

Retrieval is keyword overlap, not embeddings. Simpler, free, and honest about its limits — it finds chunks that use your question's words and can miss a paraphrase. Two details make it work: chunks overlap by 150 characters, so a sentence straddling a boundary still appears whole somewhere the model can read it; and stopwords are stripped before scoring — without that, “What is the watering schedule?” matches every chunk on *what*, *is*, and *the*, and the scoring measures grammar instead of topic. That bug appeared in the first test run of this exact code and was fixed before publication.

Storage is in memory. Restart the server and uploads are gone. Both choices are right-sized for a first app — and both return, in the tutorial's close, as the first production upgrades, alongside authentication.

Grounding, enforced by code

Only the four best-matching chunks ever reach the model — never the whole document — which keeps each question small whatever the PDF's size. And if no chunk matches, /ask answers without an API call: asking the sample handbook “What is the airspeed velocity of an unladen swallow?” returned the no-passage answer with `chunks_used`: 0 and zero tokens. Grounding enforced by code, not by hoping the prompt behaves.

The live run

The test document was a two-page community-garden handbook written as a sample. The server's real response to the upload:

```
{"doc_id": "979808d3", "filename": "sample.pdf",  
  "pages": 2, "characters": 856, "chunks": 2}
```

Cost per answer — the real figures

The live question — “When is the lottery draw, and what does a plot cost?” — came back with the handbook's own facts: the draw takes place on March 30, 2026, and each plot costs 45 dollars for the season. Answered from a single chunk, at the published gemini-3.8-flash rates (US\$0.75 per million input tokens, US\$3.75 per million output):

1 chunk used · 277 input tokens · 35 output tokens · estimated cost US\$0.000339 — roughly a thirtieth of a cent. The app prints the exact figure under every answer, so cost is a feature you can see, not a bill you discover.

PDF Q&A — the errors table

422 — “No readable text was found in that PDF.”	The file parsed but has no text layer — the classic scanned or image-only PDF. This app does not OCR; run the document through an OCR tool first.
413 — “That file is over the 10 MB limit.”	The cap is a constant at the top of app.py. Raise it deliberately if needed — retrieval, not the context window, keeps questions cheap.
400 — “Please upload a PDF file.”	The filename does not end in .pdf. A renamed text file fails here or in extraction.
404 — “Unknown document.”	The id is wrong, or the server restarted and forgot every upload — the in-memory store working as designed. Upload again.
500 — “GEMINI_API_KEY is not set on the server.”	The key is missing from the server's environment. Setting it in a different terminal does not count — set it in the window running uvicorn, then restart.
502 — “Gemini API error: ...”	The call failed — invalid key, exhausted free-tier quota, or a network problem. The text after the colon is the API's own error; fix that, not the app.
A 404 inside the 502 — “This model ... is no longer available to new users.”	Google retires model names for new keys first. If the error names a replacement model, update the MODEL constant at the top of app.py. This app pins gemini-3.8-flash for exactly this reason: a freshly created key could not reach gemini-2.5-flash at all.
The answer misses your question's meaning	Retrieval matched words, not intent — or the passage uses different words than your question. Rephrase with the document's own vocabulary, or take it as the signal to build the embeddings upgrade.

Run everything — the setup checklist

One page, all three builds. Versions are pinned exactly as the tutorials used them — pinning matters, because agent tooling changes quickly and a pinned version means the code you see is the code that runs.

Shared groundwork

- Python 3.10 or newer (the tutorials use 3.12 / 3.12.3). Check: `python --version`.
- A Gemini API key from Google AI Studio — free tier is enough, no card. `aistudio.google.com` → Get API key → Create API key → copy it. Keep it secret like a password; the code reads it only from an environment variable.
- Set it in the same terminal that runs the code: `export GEMINI_API_KEY="your-key"` (macOS/Linux) or `$env:GEMINI_API_KEY = "your-key"` (Windows PowerShell). Never paste a key into source code.
- One folder per build, one virtual environment per folder: `python3 -m venv .venv`, then activate it.

Part 1 — the agent

- `pip install google-genai==1.35.0` — one dependency, one file (`first_agent.py`), no framework.
- Save both code parts from the post into `first_agent.py` and run: `python first_agent.py`.
- You should see the step lines print (weather ×2, calculator ×1), then the answer. (Note: the post pins the model `gemini-2.5-flash`; Google has since closed that model to new keys — if your fresh key gets a “no longer available to new users” 404, point the model name at the replacement the error names, exactly as Part 3's errors table explains.)

Part 2 — the MCP server (no key, no account)

- `pip install "mcp[cli]==2.3.0"` — SDK v2; code written for v1's FastMCP will not run on it.
- Save `server.py` and `client.py` in the same folder. Inspect first: `mcp dev server.py` → List Tools → call `search_notes` with “backup”.
- Run the client: `python client.py` — expect the two Tools lines and the JSON results. The client starts the server with the same Python, so the server inherits your venv.

Part 3 — the PDF Q&A app

- `pip install fastapi==0.161.1 "uvicorn[standard]==0.54.0" pypdf==6.21.2 google-genai==2.33.0 python-multipart==0.0.32` — `python-multipart` is easy to forget, and without it FastAPI cannot accept file uploads at all.
- Verify: `python -c "import fastapi, pypdf, google.genai; print('setup ok')"`.
- Run: `uvicorn app:app --reload`, then open `http://127.0.0.1:8000`. Model pinned: `gemini-3.8-flash`.

All three builds, complete and tested, live in the companion repository — github.com/sajjad037/visual-studio-learn-code — one folder per tutorial: `topic-01-first-ai-agent` · `topic-02-mcp-server` · `topic-03-pdf-qa-app`.

Where to go next

This pack is the distillation. The full builds — every line of code, every error, the videos walking through them — live here:

The full tutorials (blog)

- Part 1 — Build Your First AI Agent in Python: visualstudiolearn.blogspot.com/2026/10/build-your-first-ai-agent-in-python.html
- Part 2 — MCP Servers Explained: visualstudiolearn.blogspot.com/2026/10/mcp-servers-explained-give-your-ai.html
- Part 3 — Small AI App #1: PDF Q&A with FastAPI + Gemini: visualstudiolearn.blogspot.com/2026/10/small-ai-app-1-build-pdf-qa-app-with.html

The videos (YouTube — @VisualStudioLearn)

- Part 1 — youtube.com/watch?v=r0nY3zq9YfE
- Part 2 — youtube.com/watch?v=SA-J4OARAzk
- Part 3 — youtube.com/watch?v=QWOKffkimuU
- Plus a one-minute Short for every topic — the complete lesson in 60 seconds — on the same channel.

The code

github.com/sajjad037/visual-studio-learn-code — the exact code from the posts, pinned requirements, and a README per topic that covers setup, install, and running.

Follow along

- Facebook Page — Visual Studio Learn: facebook.com/VisualStudio.Learn — every post and Short, shared there too.
- New tutorials every Tuesday, Thursday, and Saturday — Python, FastAPI, Microsoft Azure, SQL Server, and AI agents, in that same hands-on style.

GET NEW POSTS BY EMAIL — free, no spam. Subscribe at follow.it/follow-form/visual-studio-learn and each new tutorial lands in your inbox. Coming next in the series: an AI agent that automates a real daily task, deploying FastAPI to Azure, and more Small AI Apps — each one small, runnable, and honest about what it costs.

— *Visual Studio Learn*

Starter Pack v1 · October 2026 · Distilled from the three published tutorials above; all run outputs and figures are those tutorials' own live results.